

# Structuri de date și algoritmi

Domeniul de studii: Ingineria Sistemelor / Specializarea: Automatică și Informatică Aplicată

## SDA – Cursul 2

**Conf. dr.ing. Adriana ALBU**

[adriana.albu@upt.ro](mailto:adriana.albu@upt.ro)

<http://www.aut.upt.ro/~adrianaa>

# **1. Structuri de date fundamentale (partea întâia)**

# 1.1 Preliminarii

---

- Caracteristici ale sistemelor de calcul:
  - viteza de lucru
  - capacitatea de memorare
  - posibilitățile de acces la informațiile memorate
- Sistemele de calcul prelucrează informații
  - reprezintă o abstractizare a lumii reale
  - se concretizează într-o mulțime de date
- Constituirea informațiilor furnizate unui program comportă două etape (adeseori se întrepătrund):
  - stabilirea abstractizării valabile pentru rezolvarea problemei, în urma căreia rezultă un set de date inițial
  - stabilirea modului de reprezentare în sistem a acestor date

# 1.2 Tipuri de date / 1.2.1 Conceptul de tip de dată

---

- E necesar ca fiecare constantă, variabilă, expresie sau funcție să se încadreze unui anumit tip de dată
- Un tip de data se caracterizează prin:
  - mulțimea valorilor
  - grad (nivel) de structurare
  - set de operatori specifici
- Precizarea tipurilor de date se realizează prin **declarații**
  - explicite – preced textual utilizarea obiectelor încadrate în acele tipuri
  - implicite – unele tipuri sunt recunoscute implicit prin reprezentare

# 1.2.1 Conceptul de tip de dată

---

## ➤ Caracteristicile conceptului de tip de dată

- determină în mod univoc **mulțimea valorilor** pe care le poate asuma un element încadrat în tipul respectiv (constante, variabile sau valori generate de un operator sau o funcție)
- tipul unui element sintactic poate fi dedus din forma sa de prezentare sau din declarația sa explicită, fără a fi necesară execuția unor procese de calcul suplimentare
- fiecare operator sau funcție acceptă argumente de un tip precizat și furnizează rezultate de asemenea de un tip precizat
- presupune un anumit **grad de structurare** a informației, grad care e evidențiat de nivelul de organizare asociat tipului de dată

## ➤ **Datorită acestor caracteristici**

- compilatoarele verifică legalitatea și compatibilitatea unor construcții de limbaj, încă în faza de compilare (fără a fi necesară execuția)

# 1.2.1 Conceptul de tip de dată

---

## ➤ Metode de structurare

- prin agregare:
  - definirea unor tipuri de date noi prin agregare (conglomerare), pornind de la tipuri existente (anterior definite)
  - valorile tipurilor rezultate sunt conglomerate de valori ale tipurilor constitutive
  - dacă există un singur tip constitutiv, acesta se numește tip de bază
- prin încuibare:
  - definirea unor tipuri de date în interiorul altor tipuri

## ➤ Cele două metode pot fi combinate

- => un anumit nivel de structurare, bazat pe o ierarhie de structuri
- tipurile de la baza structurării trebuie să fie tipuri primitive (atomi)

# 1.2.1 Conceptul de tip de dată

---

- Tipuri primitive nestructurate (fundamentale):
  - tipuri standard, numite și **predefinite**, care au de regulă corespondență cu reprezentări asociate arhitecturii hardware a sistemului de calcul
  - **definite de utilizator** prin enumerarea valorilor constitutive ale tipului
- Dacă între valorile individuale ale tipului există o relație de ordonare, atunci tipul de dată este ordonat sau **scalar**
  - majoritatea tipurilor primitive sunt scalare
- Metodele de structurare de bază generează tipuri de date structurate:
  - statice
  - dinamice
  - (sunt definite de utilizator)

## 1.2.2 Conceptul de tip de data abstract (TDA)

---

➤ Un TDA poate fi definit drept o asociere între:

- un model matematic (MM)
- un set de operatori definit pentru valorile manipulate de modelul matematic

➤ Un TDA

- generalizează noțiunea de tip de dată primitiv
- încapsulează elementele specifice definiției sale
  - încapsulează alături de structura de date corespunzătoare modelului matematic și procedurile sau funcțiile care materializează operatorii

➤ La rândul său, o procedură sau o funcție

- este un element fundamental de programare care:
  - generalizează noțiunea de operator
  - încapsulează elementele specifice operatorului definit prin procedură sau funcție
    - încapsularea are ca rezultat definirea procedurii sau funcției într-un singur loc în program (încapsulare cod)



## 1.2.2 Conceptul de tip de data abstract (TDA)

---

- MM                                      <-> structura de date
- Set de operatori                      <-> proceduri sau funcții
  
- În urma definirii, un TDA poate fi tratat în continuare ca un tip primitiv de dată
  - utilizatorul nemaifiind preocupat de maniera de proiectare a tipului de data, ci doar de maniera de acces (interfață)
- TDA permit de asemenea o abordare ierarhică în construirea tipurilor structurate abstracte (ex. TDA lista)

## 1.2.2 Conceptul de tip de data abstract (TDA)

---

### ➤ Avantajele utilizării TDA

- În urma definirii unui TDA, toate prelucrările se vor realiza în termenii operatorilor definiți asupra acestui tip
- Indiferent de maniera concretă de implementare a TDA, forma operatorilor, respectiv **interfața** acestora cu utilizatorul **rămâne nemodificată**
  - permite modificarea cu ușurință a implementării cu condiția păstrării prototipului funcției asociată operatorului
- Nu există nicio limită asupra numărului de operații care pot fi aplicate instanțelor unui model matematic precizat
  - cu specificația că fiecare set de operații definește și se refera la un TDA distinct

## 1.2.2 Conceptul de tip de data abstract (TDA)

---

### ➤ Implementarea unui TDA

- Prin implementarea unui TDA se înțelege translatarea (exprimarea) sa în termenii unui limbaj de programare:
  - definirea structurii de date care materializează modelul matematic al TDA
  - scrierea procedurilor (funcțiilor) care implementează operatorii precizați în definirea TDA (utilizând noțiunile limbajului de programare sau operatorii definiți deja)
- Un TDA se implementează pornind de la tipurile de bază specifice limbajului de programare și utilizând facilitățile de structurare puse la dispoziție de limbaj

### ➤ Prin implementare un TDA devine tip de data (TD)

## 1.2.2 Conceptul de tip de data abstract (TDA)

---

- TD sunt utilizate pentru declararea unor variabile de tipul respectiv
  - aceste variabile se numesc **instanțe** ale tipului de dată
  - procesul de declarare a variabilelor se numește **instanțiere**
- Variabila – un element sintactic precizat printr-un nume simbolic
  - i se asociază o **zonă de memorie** (de o anumită dimensiune, specifică tipului de dată), al cărei conținut îl reprezintă valoarea curentă a variabilei
- Structura de date – o colecție specifică de variabile, aparținând unuia sau mai multor tipuri de date, asociate în diverse moduri, cu scopul de a facilita accesul, prelucrarea și relațiile dintre entitățile constitutive

# Rezumat pentru 1.2

**Tip de date abstract (TDA)**  
asociere între un model matematic (MM) și un set de operatori specifici

**Tip de date (TD)**  
implementare a unui TDA într-un limbaj de programare

**se caracterizează prin:**

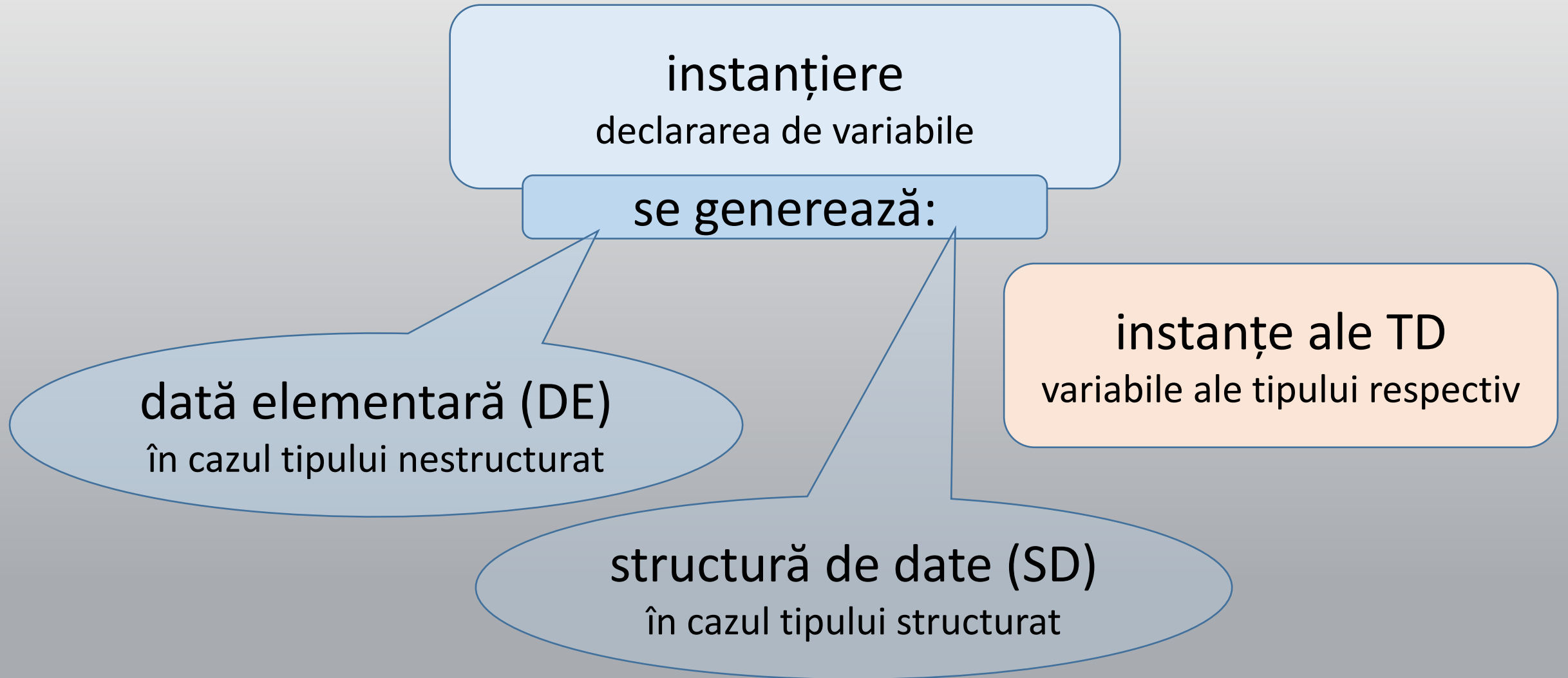
- mulțimea valorilor (pe care le pot lua elementele tipului respectiv)
- un anumit grad (nivel) de structurare (organizare) a informației
- set de operatori specifici

**pot fi:**

- nestructurate
- structurate

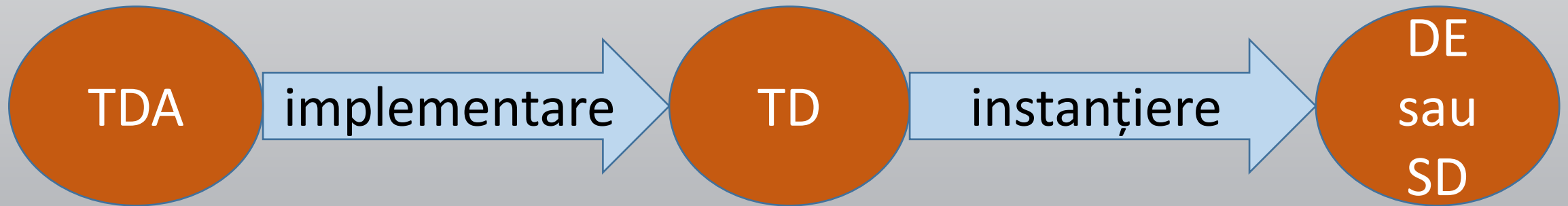
# Rezumat pentru 1.2

---



# Rezumat pentru 1.2

---



## 1.3 Tipuri primitive nestructurate /

### 1.3.1 Tipuri primitive standard (predefinite)

- Fac parte din categoria tipurilor nestructurate
- Sunt numite standard deoarece sunt implementate direct prin caracteristicile hardware ale sistemului
  - întreg
  - real
  - boolean
  - caracter
- TDA întreg, real, boolean, caracter sunt definite prin limbaj, pentru utilizare fiind necesară doar instanțierea (se încadrează în categoria tipurilor de date)



# 1.3.1 Tipuri primitive standard (predefinite)

---

## ➤ Tipul întreg

➤ este format din submulțimea reprezentabilă a numerelor întregi

## ➤ operatori:

- atribuire
- comparare
- aritmetici (+, -, x, /, %)
- operatorul DIV – împărțire întreagă, cu ignorarea restului
  - $(m \text{ DIV } n) \times n \leq m$
- operatorul MOD – modulo, furnizează restul împărțirii întregi
  - $(m \text{ DIV } n) \times n + (m \text{ MOD } n) = m$
- Uzual sunt mai multe categorii de tipuri întregi (signed, unsigned, short, long)

# 1.3.1 Tipuri primitive standard (predefinite)

## ➤ Tipul întreg

- MM – o mulțime de elemente scalare cu valori în mulțimea numerelor întregi ex.:  $\{-20, \dots, -2, -1, 0, 1, 2, \dots, 20\}$

- Notatii

- i, j, k – variabile întregi
- inz – întreg diferit de zero
- inn – întreg non-negativ
- e – valoare întreagă
- b – valoare booleană

### Operatori:

AtribuireIntregi(i,e)  
AdunareIntregi (i,j) -> k  
ScadereIntregi(i,j) -> k  
InmultireIntregi(i,j) -> k  
ImpartireIntregi(i,inz) -> k  
Modulo(i,inz) -> inn  
EgalZero(i) -> b  
MaiMareCaZero(i) -> b

- Într-un limbaj de programare implementarea acestor operatori se realizează în mod direct

## 1.3.1 Tipuri primitive standard (predefinite)

---

### ➤ Tipul real

- Implementează o submulțime reprezentabilă a numerelor reale
- Operatori: +, -, x, /
- Calculele se realizează cu o anumită **aproximație**, funcție de lungimea zonei de memorie utilizată în reprezentarea valorilor reale (rotunjiri cauzate de efectuarea calculelor cu un număr finit de zecimale)
- Implementarea operatorilor se realizează direct prin construcții sintactice ale limbajelor de programare
- Uzual sunt mai multe categorii de tipuri reale (float, double)

# 1.3.1 Tipuri primitive standard (predefinite)

---

## ➤ Tipul boolean

- Implementează valorile logice standard Adevărat și Fals
- În limbajul C acest tip lipsește fiind substituit de valorile întregi:
  - 1 sau diferit de zero – adevărat
  - 0 – fals
- Operatori specifici: operatorii logici
  - conjuncție – AND, &&
  - reuniune – OR, ||
  - negație – NOT, !
- Operatorul **comparație** aplicat asupra valorilor oricărui tip de date conduce la rezultate de tip boolean
  - poate fi atribuit unei variabile de tip logic
  - poate fi utilizat ca operand al unui operator logic într-o expresie booleană

# 1.3.1 Tipuri primitive standard (predefinite)

---

## ➤ Tipul caracter

- Mulțimea de caractere reprezentabile de regulă la un dispozitiv de afișare (mulțimea valorilor depinde de dispozitivul de ieșire)
- Au fost stabilite diverse convenții de codificare a caracterelor:
  - ISO ( International Standard Organisation)
  - ASCII (American Standard Code for Information Interchange)
- Uzual se utilizează valori întregi fără semn în domeniul [0, 255], reprezentabile pe un octet
- Codurile sunt grupate (submulțimi ordonate și coerente) pe litere mari, mici, cifre, semne de punctuație, alte caractere reprezentabile etc.
  - principiul de codificare permite determinarea apartenenței unui caracter la una dintre submulțimi
- Unele limbaje de programare pun la dispoziție funcții de transfer între tipul caracter și alte tipuri (în C: `atoi()`, `atof()`, `itoa()`)

## 1.3 Tipuri primitive nestructurate / 1.3.2 Tipul enumerare

---

- Oferă o manieră de a utiliza în cadrul programelor numere întregi în locul noțiunilor abstracte
- Este un tip definit de utilizator prin enumerarea tuturor componentelor sale
- $\text{card}(T)$ 
  - cardinalitatea tipului  $T$  – numărul valorilor distincte aparținând unui tip de dată  $T$
  - uzual cardinalitatea este utilizată drept o măsură a cantității de memorie necesară reprezentării tipului  $T$
- Definirea unui tip enumerare, introduce nu numai un nou identificator de tip, ci în același timp introduce identificatorii (simbolurile) care precizează valorile noului tip

# 1.3 Tipuri primitive nestructurate / 1.3.2 Tipul enumerare

## ➤ enum

```
enum day {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
```

- tip scalar
- constante sunt asimilate cu numerele întregi atribuite succesiv în timpul procesului de declarare a tipului
  - excepție: situația în care utilizatorul face atribuire explicit

```
enum day today, tomorrow;
```

```
enum month {Jan=31, Feb=28, Mar=31, Apr=30, May, Jun=30, Jul, Aug=31, Sep=30, Oct, Nov=30, Dec};
```

## ➤ Operatori:

- atribuire
- comparație
- succesori, predecesori
- transformare (forțare) de tip – conversia uzuală este cea între tipul enumerare și întreg

```
today=Sunday;
```

```
if (today==Sunday) tomorrow=Monday;  
else tomorrow=today+1;
```

## 1.3 Tipuri primitive nestructurate / 1.3.3 Tipul subdomeniu

---

- Precizează un domeniu restrâns în care poate lua valori un anumit tip
- Se definește în raport cu un tip deja existent, numit tip scalar asociat

TYPE TipSubdomeniu = min..max;

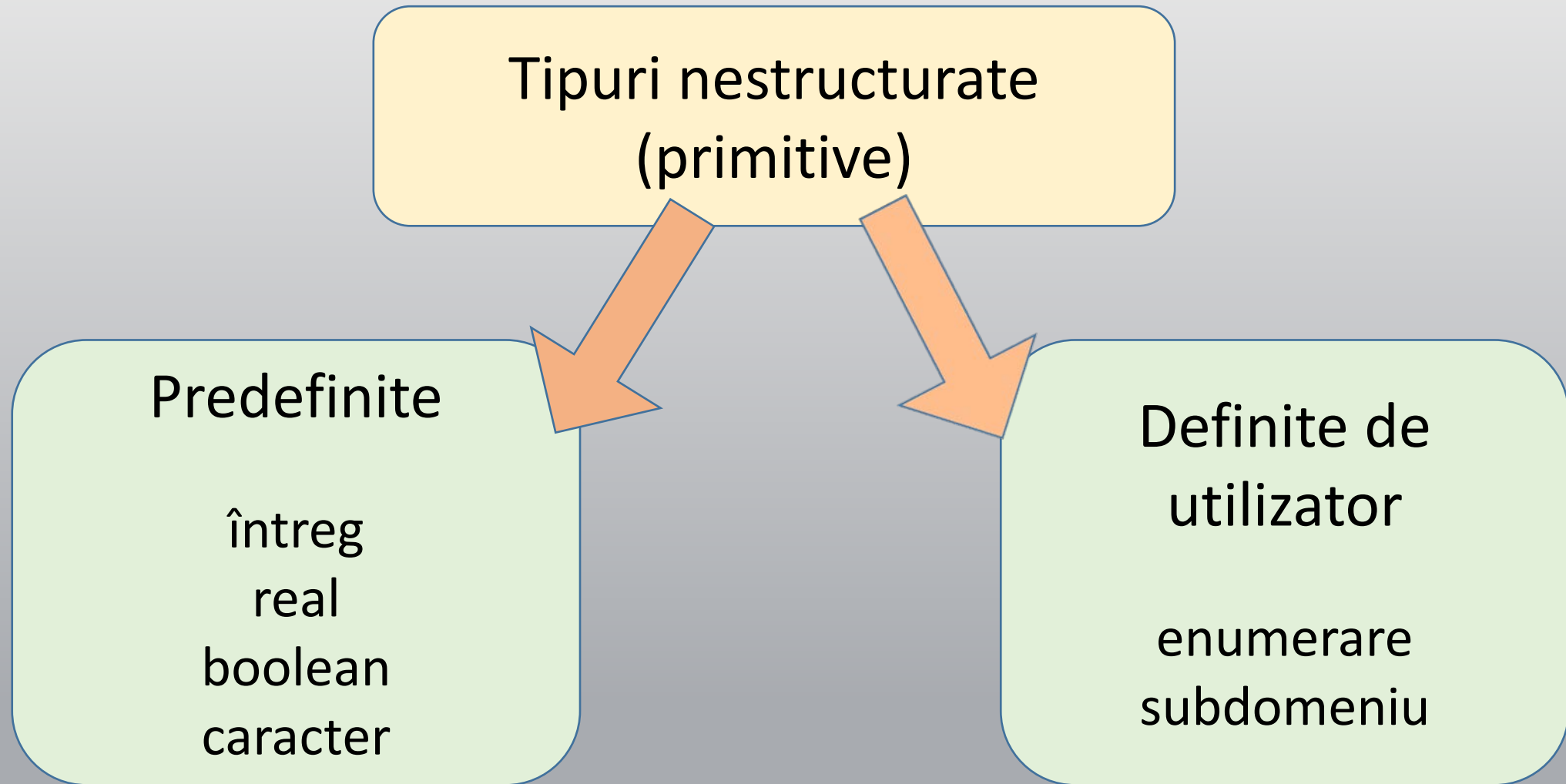
TYPE AN=2020..2024;

- Implementarea tipului subdomeniu impune verificarea de către compilator a legalității atribuirilor; în cazul atribuirii unor variabile, verificările se produc doar în timpul execuției programului. Din acest motiv scade eficiența codului generat, dar crește siguranța execuției.
- (Nu e specific limbajului C)



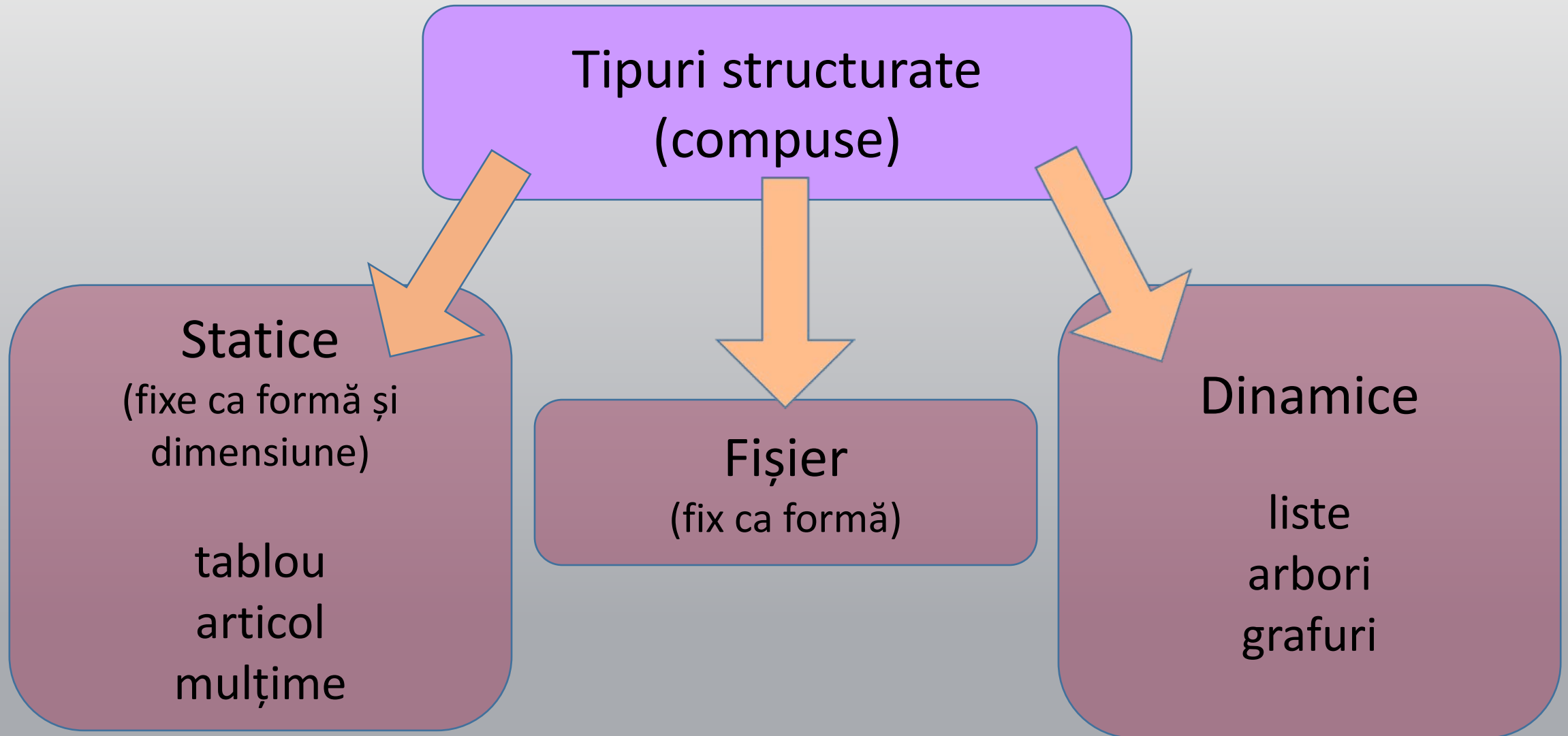
# Rezumat pentru 1.3

---



# Intro pentru 1.4

---



## 1.4 Tipuri structurate / 1.4.1 Structura tablou. TDA tablou

---

- Construcția se bazează pe facilitățile hard ale sistemului de calcul, respectiv pe **mecanismul de adresare indexată**
- Structură de date **omogenă** ale cărei componente aparțin aceluiași tip constitutiv, numit **tip de bază**
- Structură cu **acces direct** (random acces)
  - toate elementele sale sunt direct și în mod egal accesibile
- Structură **statică**, cu dimensiune și formă fixă
- **Indicii de acces** la elementele tabloului aparțin unui tip scalar
  - indicii selectează pozițional componenta dorită
- Operatori: constructor, selector, atribuire

# TDA tablou

---

## ➤ MM

- secvență de elemente de același tip
- indicele asociat fiecărei componente aparține unui tip ordinal finit
- există o corespondență biunivocă între valoarea indicelui și componentele tabloului (corespondență unică în ambele sensuri)

## ➤ Notatii:

- TipElement a – tablou unidimensional;
- i – TipIndice;
- e – obiect de TipElement

## ➤ Operatori:

- DepuneTablou (a, i, e); // a[i]=e;
- TipElement FurnizeazaDinTablou (a, i); // e=a[i];

## 1.4.2 Tehnici de căutare în tablouri

---

### ➤ Căutarea (regăsirea informației)

- operație frecventă
- => una dintre tehnicile cele mai abordate și studiate în programare
- viteza cu care se efectuează această operație într-o bază de date de dimensiuni mari **influențează în mare măsură eficiența** unei aplicații
- presupune efectuarea unor operații de **comparație**
- => uneori elementele tabloului se identifică printr-o **cheie**, ce aparține unui tip scalar (poate fi constituită dintr-un singur câmp sau o combinație de câmpuri)
- uzual, returnează indicele elementului din tablou pentru care cheia coincide cu cheia de căutare dată, respectiv indicele elementului a cărui valoare coincide cu valoarea de căutat dată, în cazul în care nu se utilizează o cheie pentru identificare

## 1.4.2 a) Căutarea liniară

---

- Parcurgerea secvențială (traversarea) a tabloului
  - unica metodă de căutare dacă nu avem nicio informație apriorică asupra tabloului
  - realizată într-o manieră ordonată prin incrementarea (decrementarea) indicelui
- Căutarea elementului  $x$  într-un tablou unidimensional  $a$
- Rezultatul:
  - găsirea elementului de valoare  $x$  și eventuala returnare a indicelui ( $a[i]==x$ )
  - negăsirea elementului  $x$  și depășirea tabloului ( $i>N-1$  dacă indicii aparțin  $0...N-1$ )

# Căutarea liniară

---

```
int i=0;
while( (i<N)&&(a[i]!=x) ) i++;
if(a[i]!=x)
    printf("Nu exista elementul cautat");
else
    printf("Elem. cautat este pe poz. %d", i);
```

➤ **invariant:** condiția care dacă este îndeplinită, permite rămânerea în buclă (respectiv reluarea buclei)

- $(0 \leq i < N) \wedge (a[i] \neq x)$

➤ **condiția de terminare:**

- $((i == N) \vee (a[i] == x))$

# Aprecierea performanțelor căutării liniare

---

- Criteriul: numărul de comparații efectuate până la găsirea elementului căutat
- Caz particular:
  - $x$  pe poziția  $i$   $\rightarrow$  se efectuează  $2 * (i + 1)$  comparații
  - timpul de execuție proporțional cu  $i$
- Cel mai defavorabil caz:
  - $x$  pe poziția  $N - 1$   $\rightarrow 2 * N$  comparații
- În cazul în care elementul  $x$  se găsește cu siguranță în tablou, numărul de comparații cel mai probabil este  $2 * N / 2$  (apreciere timp mediu)
- **Performanța căutării liniare:** în medie un element este găsit după parcurgerea a jumătate dintre elementele tabloului
- Reducerea timpului de execuție se poate obține prin reducerea numărului de operații efectuate în fiecare ciclu



## 1.4.2 b) Căutarea liniară. Metoda fanionului

---

- Simplificarea invariantului se poate realiza garantând cel puțin o „potrivire” în procesul de căutare (prezența elementului  $x$  în tablou)
- Se completează tabloul cu o locație suplimentară, în care se plasează elementul căutat  $x$

...

```
a[N]=x;
```

```
i=0;
```

```
while(a[i]!= x) i++;
```

```
if(i==N)
```

```
    printf("nu exista elementul cautat");
```

```
else
```

```
    printf("Elem. cautat este pe poz. %d", i);
```

...

# Căutarea liniară. Metoda fanionului

---

## ➤ Elimină

- $n$  teste în caz de căutare fără succes
- $i$  teste în caz de căutare cu succes (reducere a timpului cu 20-50%)

## ➤ Alte variante de reducere a timpului în căutările liniare:

- păstrarea datelor ordonate după cheie
  - permite abandonarea căutării la un moment dat
- păstrarea datelor ordonate după frecvența cu care sunt căutate
  - timpul mediu se reduce simțitor

## 1.4.2 c) Căutarea binară

---

- Informații suplimentare referitoare la organizarea datelor prezente în tablou => accelerarea căutării
- Elementele tabloului sunt ordonate conform unui anumit criteriu:
  - $A_k : 0 < k < N : a_{k-1} \leq a_k$
- Principiul căutării binare:
  - se compară elementul căutat  $x$  cu elementul aleatoriu  $a_m$
  - dacă  $a_m < x$ , căutarea continuă în intervalul care conține elemente cu indici mai mari decât  $m$
  - dacă  $a_m > x$ , căutarea continuă în intervalul care conține elemente cu indici mai mici decât  $m$
  - dacă  $a_m = x$ , elementul a fost găsit și căutarea se oprește
  - procesul continuă până când se găsește elementul  $x$  sau intervalul în care se execută căutarea devine vid

# Căutarea binară

➤ indicii  $s$  și  $d$  precizează limita stângă și dreaptă a intervalului

```
int s=0, d=N-1, gasit=0, m;  
while( (s<=d)&&(!gasit) ) {  
    m=(s+d)/2; //orice valoare cuprinsa intre s si d  
    if(a[m]==x) gasit=1;  
    else if(a[m]<x) s=m+1;  
        else d=m-1;  
}
```

➤ Invariant:  $(s \leq d) \ \&\& \ (A_k : 0 \leq k < s : a_k < x) \ \&\& \ (A_k : d < k < N : a_k > x)$

➤ Eficiența algoritmului este influențată de alegerea lui  $m$ :

- scopul urmărit este de a reduce în fiecare pas intervalul în care se face căutarea

# Căutarea binară

---

- Dacă fiecare intrare în buclă asigură înjumătățirea intervalului:
  - după prima trecere vor rămâne  $N/2$  elemente de procesat
  - după  $k$  treceri vor rămâne  $N/2^k$  elemente de procesat
- Procesul continuă până la un interval care conține 1 element

$$N/2^k < 1 \quad \rightarrow \quad N < 2^k \quad \rightarrow \quad \log_2 N < k \text{ (notația } \lg N \text{)}$$

- Numărul maxim de treceri:  $\log_2 N$  (rotunjit superior)
- Căutarea binară, numită și bisecție, are la bază procesul eliminării:
  - ori de câte ori se face o comparație în procesul de căutare, sunt posibile două soluții pentru alegerea noului interval de căutare

## 1.4.2 d) Căutarea binară performată

➤ Simplificarea invariantului prin renunțarea la condiția „!gasit”

```
int s=0, d=N, m;
```

```
while(s<d){
```

```
    m=(s+d)/2;
```

```
    if(a[m]<x) s=m+1;
```

```
    else d=m;
```

```
}
```

```
if(d==N) printf ("elementul nu exista")
```

```
if(d<N)
```

```
    if(a[d]==x) printf("elem exista");
```

```
    else printf("elem nu exista");
```

➤ Găsește elementul  $x$  cu cel mai mic indice

## 1.4.2 e) Căutarea prin interpolare

---

### ➤ Creșterea performanței:

- determinarea cu mai multă precizie a intervalului în care se face căutarea
- căutare binară:  $m = (s+d)/2 = s+(d-s)/2$
- căutare prin interpolare:  $m = s + (d-s) (x-a[s]) / (a[d]-a[s])$

### ➤ În cazul unei distribuții uniforme a elementelor, $(x-a[s]) / (a[d]-a[s])$ va accelera procesul de căutare;

- numărul de treceri estimat:  $\lg(\lg(N))$

### ➤ Dezavantaje:

- depinde de distribuția elementelor; orice abatere afectează performanța
- este performantă pentru valori foarte mari ale lui N

### ➤ Avantaje:

- se utilizează când elementele tabloului sunt structuri complexe
- pentru căutare externă, care necesită costuri suplimentare de acces

**Vă mulțumesc!**